

# Patmos: a time-predictable microprocessor

**Schoeberl, Martin; Puffitsch, Wolfgang; Hepp, Stefan; Huber, Benedikt; Prokesch, Daniel**

*Published in:*
Real-Time Systems

*Link to article, DOI:*
10.1007/s11241-018-9300-4

*Publication date:*
2018

*Document Version*
Peer reviewed version

**Noname manuscript No.**
(will be inserted by the editor)

# Patmos: A Time-predictable Microprocessor

**Martin Schoeberl · Wolfgang Puffitsch · Stefan Hepp · Benedikt Huber · Daniel Prokesch**

Received: date / Accepted: date

**Abstract** Current processors provide high average-case performance, as they are optimized for general purpose computing. However, those optimizations often lead to a high worst-case execution time (WCET). WCET analysis tools model the architectural features that increase average-case performance. To keep analysis complexity manageable, those models need to abstract from implementation details. This abstraction further increases the WCET bound.

This paper presents a way out of this dilemma: a processor designed for real-time systems. We design and optimize a processor, called Patmos, for low WCET bounds rather than for high average-case performance. Patmos is a dual-issue, statically scheduled RISC processor. A method cache serves as the cache for the instructions and a split cache organization simplifies the WCET analysis of the data cache. To fill the dual-issue pipeline with enough useful instructions, Patmos relies on a customized compiler. The compiler also plays a central role in optimizing the application for the WCET instead of average-case performance.

**Keywords** Real-time systems · Time-predictable architecture · Worst-case execution time

## 1 Introduction

Analyzing the timing properties of an application is of major importance for real-time systems. Only a time-predictable platform can enable the computation of safe yet tight bounds on the worst-case execution time (WCET). Therefore, real-time systems need time-predictable processors.

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
E-mail: masca@dtu.dk

This paper presents a time-predictable processor, Patmos, which is optimized for real-time systems. Its design enables the computation of tight WCET bounds, while at the same time providing good performance. First concepts of Patmos have been presented in prior work [62].

The contributions of this paper and the Patmos project in general are:

1. The design and implementation of a time-predictable processor
2. A compiler supporting the processor
3. A WCET analysis tool for Patmos
4. The evaluation of the performance and size of Patmos

Processors for future embedded systems need to be time-predictable *and* provide reasonable worst-case performance. We present the time-predictable processor Patmos, which simplifies the usually complex WCET analysis. Patmos is a statically scheduled, dual-issue RISC processor that is optimized for real-time systems. Instruction delays are well defined and visible through the instruction set architecture (ISA). This design simplifies the WCET analysis and helps to reduce the overestimation caused by imprecise information.

The dual-issue pipeline, with specially designed caches, provides good single-thread performance. The chip-multicore version of Patmos provides a time-predictable execution platform for multi-threaded applications.

A major challenge for the WCET analysis is the memory hierarchy with multiple levels of caches. We tackle this issue through caches especially designed for WCET analysis. For instructions, we adopt the method cache [51], which operates on entire functions/methods and thus simplifies the modeling for WCET analysis. Furthermore, we propose a split cache architecture for data [53, 60], offering dedicated caches for the stack [1] and for other data. Patmos also supports scratchpad memories for instructions and data.

In addition to the hardware implementation of Patmos, we also present a compiler for the development of real-time applications. As Patmos is designed to facilitate WCET analysis, its internal operation is thus well defined in terms of timing behavior and explicitly made visible on the ISA level. Features that are hard to predict are avoided, instead replaced by more predictable alternatives, some of which rely on the (low-level) programmer or compiler for achieving optimal results, i.e., low actual WCET and good WCET bounds. We provide a *WCET-aware* software development environment that tightly integrates WCET tools and the compiler [48,9].

The processor and its software environment are intended as a platform for exploring various time-predictable design trade-offs and their interaction with WCET analysis techniques and WCET-aware compilation. We propose the co-design of time-predictable processor features with the WCET analysis tool, like the work by Huber et al. [27] on the caching of heap-allocated objects in a Java processor. Only features for which we can provide a static program analysis will be added to the processor.

Patmos and its tool chain are open source; the hardware implementation is available under the BSD license.[1] Detailed descriptions of the instruction set and the build process are available in the Patmos handbook [57].

The presented processor is named after the Greek island Patmos, where the first sketches of the architecture have been drawn; not in sand, but in a (paper) notebook. If you use the open-source design of Patmos for further research, we would suggest that you visit and enjoy the island Patmos. Consider writing a postcard from there to the authors of this paper.

This paper is organized in 8 sections: The following section, Section 2, presents related work. Section 3 provides background information on the T-CREST project, where Patmos is the processing node for a multicore processor. Section 4 presents the design of Patmos. Section 5 describes our adaption of the LLVM compiler for Patmos and the integration with WCET analysis. Section 6 presents the concrete implementation of Patmos in an FPGA. Section 7 evaluates the design. Section 8 concludes.

## 2 Related Work

Digital hardware is in principle perfectly predictable: with the same hardware start state and the same input data, two runs of a program will result in the same result and execution time. However, the unknown initial hardware state makes WCET analysis for modern processors very difficult. To solve this problem, several research groups have started to investigate predictable processors and memory hierarchies.

Thiele and Wilhelm argue that a new research discipline is needed for time-predictable embedded systems [66]. Edwards and Lee state: "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [14]. In line with these arguments, we consider Patmos a design that implements a time-predictable processor.

The focus of the precision timed (PRET) machine [14] is primarily on repeatable timing. However, repeatable timing with predictable programming can also lead to time predictability. A deadline instruction can be used for enforcing the repeatable timing of a task. A first simulation of the PRET architecture is presented by Lickly et al. [35]. The first hardware implementation of PRET implements the ARM instruction set [38, 37]. PRET implements a RISC pipeline and performs chip-level multithreading for four threads to eliminate data forwarding and branch prediction [39]. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a time-division multiplexing (TDM) scheme, called "memory wheel."

PRET [13] ensures time-predictable access to SDRAM by assigning each thread a dedicated bank in the memory chips. The access to the individual banks is pipelined, and the access time is fixed. As the memory banks are not

[1] see: `https://github.com/t-crest/patmos`

shared between threads, thread communication must be performed via the shared scratchpad memory.

A recent version of PRET, FlexPRET [73], extends PRET to support mixed-criticality systems. FlexPRET supports two different thread types, hard and soft real-time threads, directly in the hardware. Both thread types have fixed slots assigned in the fine-grained thread scheduling. However, slots not used by a thread (e.g., because of stalling or because a thread has finished its current release) can be used by the soft real-time threads. FlexPRET implements the RISC V instruction set.

In contrast to the PRET approach, we use a dual-issue pipeline for maximum single-thread performance. For multi-threaded applications, we provide a multicore version of Patmos.

Within the FP-7 project MERASA (Multi-Core Execution of Hard Real-Time Applications Supporting Analysability) [67], the real-time processor CarCore [42] was developed. CarCore is a simultaneous multi-threading version of the TriCore processor. The CarCore is a two-way, five-stage pipeline with separate address and data paths. This architecture allows that the fetch stage issues an address-related instruction (load/store) and an integer instruction within one cycle. CarCore supports a single hard real-time thread to be executed with several non-real-time threads running concurrently in the background. The real-time thread uses a dynamic instruction scratchpad [41] that caches full functions. This approach is like the method cache [51] used in JOP [52] and in Patmos. Stack-allocated data is stored in a data scratchpad memory. The non-real-time threads are served by conventional instruction and data caches.

We share the same vision as the MERASA project: building hardware to support real-time systems and WCET analysis. In contrast to the MERASA processor core, Patmos focuses on single-thread real-time performance. To benefit from thread-level parallelism, we replicate the simple pipeline to build a chip-multiprocessor system. For time-predictable multi-threading, almost all resources (e.g., thread-local caches and register files) need duplication. Therefore, we believe that a multicore system is more efficient than simultaneous multi-threading.

The SPEAR (Scalable Processor for Embedded Applications in Real-time Environments) processor was designed to be time-predictable. SPEAR avoids caches and has a constant execution time for each instruction [12]. SPEAR supports single-path programming with a conditional move. In contrast to SPEAR, Patmos contains time-predictable caches and supports single-path programming with a fully predicated instruction set.

The JOP project explored time-predictable architectures within the context of a Java processor [52]. The pipeline and the microcode have been designed to avoid timing dependencies between bytecode instructions. JOP uses split load instructions to partially hide memory latencies. Caches are designed to be time-predictable and analyzable [51, 53, 61, 27, 60]. With Patmos we build on our experience with JOP and implemented a similar, but more general, cache structure.

The FP-7 project JEOPARD (Java Environment for Parallel Realtime Development) investigated architectures and tools for real-time Java on multicore systems. Within the hardware architecture work package, JOP was extended to support time-predictable execution of Java applications on a multicore. The TDM-based memory access arbitration [43] was incorporated into JOP's WCET analysis tool. For Patmos we extended the idea of TDM-based memory arbitration to a distributed TDM memory arbiter [59].

Whitham argues that the execution time of a basic block must be independent of the execution history [68]. To reduce the WCET, Whitham proposes implementing the time critical functions in microcode on a reconfigurable function unit. Whitham and Audsley extend the MCGREP architecture with a trace scratchpad [69], which caches microcode and is placed after the decode stage. A software tool extracts instruction-level parallelism at the microcode level and schedules the instructions statically.

Starke, Carminati, and Oliveira present a VLIW processor for real-time systems [65,64]. The processor implements a four-issue pipeline, supports static branch prediction, and predication. For the memory hierarchy, the processor uses a direct mapped instruction cache and a scratchpad memory for data. The processor has been implemented in a Cyclone IV FPGA, consumes 21,220 logic cells and can be clocked at 93 MHz. The processor is in the same spirit as the Patmos approach by being designed for real-time system using a simple pipeline, but supporting multiple instruction issue.

Falk et al. [15,16] developed the WCET-driven compiler WCC, guided by the results of the aiT WCET analysis tool [22] WCC optimizes the worst-case path. With our LLVM-based compiler, we also have a tight integration with the aiT WCET analysis tool.

General suggestions for future architectures of memory hierarchies are given by Wilhelm et al. [71]. A collection of features for time-predictable systems starting from the hardware, suggesting synchronous programming languages, and presenting WCET-aware compilation are presented by Axer et al. [5].

A WCET-predictable super-scalar processor that includes a mechanism to avoid long timing effects is proposed by Rochange and Sainrat [50]. Their idea is to restrict the fetch stage to fetch multiple instructions only from the same basic block. For the detection of basic blocks in the hardware, additional compiler-inserted branches or special instructions are suggested.

## 3 T-CREST: Time-predictable Multicore Architecture for Embedded Systems

The processor Patmos we present in this paper is one component in the T-CREST[2] project [56], funded by the European Commission. Its goal is to develop a time-predictable multicore processor. The project aims at developing time-predictable solutions for a processor core, the on-chip interconnect, the

[2] see `http://www.t-crest.org/`

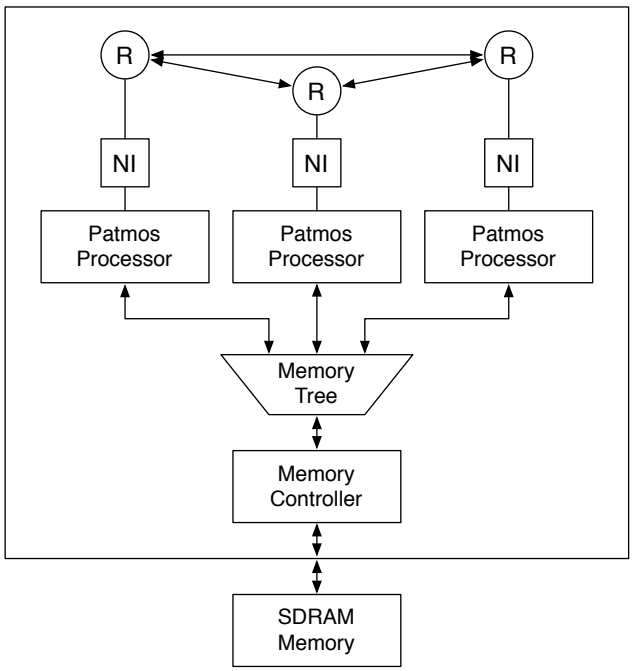


**Fig. 1** The T-CREST platform consisting of Patmos processor nodes that are connected via a NoC for message passing and a memory tree for shared memory access.

memory hierarchy, and the compiler and WCET analysis. This section provides background information on T-CREST and gives an overview of the vision of the T-CREST project.

The T-CREST approach is to design computer architectures where predictable timing is a first-order design factor [55]. For real-time systems, we thus propose to design architectures with a new paradigm [54]:

Make the worst-case fast and the whole system easy to analyze.

Time-predictable caching and time-predictable chip-multiprocessing provide a solution for the increased processing power needs in the real-time domain. T-CREST covers technologies at different levels: (1) the chip level design (processor, memory, asynchronous network-on-chip), (2) compiler technologies with WCET optimization and single-path code generation, (3) WCET analysis tools, and (4) system evaluation. For the system evaluation, T-CREST includes a port of a real-time operating system and two industry use cases, one from the avionics domain and one from the railway domain.

Figure 1 shows the T-CREST platform. Several Patmos processors are connected via a memory tree [19] to a real-time memory controller [2,33,20]. The memory controller is connected to an external SDRAM memory. Therefore, main memory is shared between processor cores. For efficient core-to-core communication, each processor is connected to a time-predictable network-on-chip (NoC) [58]. A Patmos processor is connected to a network interface (NI).

Each NI is itself connected to a router (R). The routers are connected to their neighbor routers. These on-chip communication channels reduce the pressure on the shared memory bandwidth.

Most of the T-CREST hardware is open-source under the industry-friendly simplified BSD license. The build instructions for the whole platform can be found at `https://github.com/t-crest/patmos` and in more detail in the Patmos handbook [57].

### 3.1 The Interconnect

To build a chip-multiprocessor system out of Patmos processor cores, we need a suitable interconnect—a network-on-chip (NoC). The Patmos multiprocessor platform includes distributed local memories that are connected to each processor. The NoC supports time-predictable data movement between these local memories.

To enable time-predictable usage of a shared resource, the resource arbitration must be time-predictable. For a NoC, statically scheduled time-division multiplexing (TDM) is a time-predictable solution [58]. This static schedule is repeated, and the length of the schedule is called the period. Like tasks in real-time systems, the communication is also organized in periods. The T-CREST NoC uses TDM from end to end, including the network interface. This approach also results in an efficient implementation of the network interface [63]. The latest T-CREST NoC, which is called Argo, uses a globally asynchronous locally synchronous hardware design with asynchronous routers [30].

### 3.2 Memory Hierarchy

Cache memories for instructions and data are classic examples of the paradigm "make the common case fast." A great deal of effort has gone into researching the integration of the instruction cache into the timing analysis [4] and the integration of the cache analysis with the pipeline analysis [21]. The influence of different cache architectures on WCET analysis is described by Heckmann et al. [23]. Within T-CREST we explore time-predictable caches for Patmos. Furthermore, we also consider the integration of program- or compiler-managed scratchpad memories (SPMs).

Even for embedded systems, the on-chip available memory is usually too small to hold all code and data. Therefore, an off-chip SDRAM serves as a shared main memory for the multicore processor. Access time to the SDRAM depends on the history of former accesses (e.g., open rows). This optimization improves the average case execution time—but not the WCET. Therefore, the T-CREST team developed real-time memory controllers [2,33,20]. The cores are connected to the memory controller through a memory tree, known as the "Bluetree" [19]. The Bluetree communication tree is built out of two input multiplexers with arbitration. A tree out of these two input circuits connects

all processors to a single memory controller. Arbitration is priority based, i.e., the left input has priority over the right input. That means processors have different priorities towards the memory, depending on their position. To avoid starvation, the left channel is only allowed an upper bound of requests before the right channel is served. In effect, when the system is light loaded this tree structure allows quick service and on heavy load there is an upper bound on the waiting time till service. The Bluetree avoids unused slots as would emerge on TDM arbitration when one processor has no outstanding memory request.

Even on a multicore system, the combination of the time-predictable memory tree [19, 59] and the time-predictable memory controller allows us to provide upper bounds on memory transactions. This upper bound enables the WCET analysis of individual tasks being executed on a multicore system.

### 3.3 Compiler and WCET Analysis

In addition to the hardware, the T-CREST team has developed a compiler infrastructure. The compiler for Patmos is an adaption of the LLVM compiler infrastructure [48, 26]. The time-predictable processor Patmos, with the available timing model, allows the development of WCET-aware optimization methods.

Within the project two WCET analysis tools that support Patmos have been used. The commercial WCET analysis tool aiT [22] from AbsInt has been adapted to support Patmos. The WCET-oriented optimization in the compiler is tightly integrated with the WCET analysis tool [48], which provides information on the worst-case path and basic block timings for guiding the optimization process. Furthermore, a tool called `platin` has been developed. The role of `platin` is twofold: (1) being the interface between the compiler and the aiT tool and (2) providing a set of analyses and transformations for WCET analysis itself.

### 3.4 Operating Systems for Patmos

Although not directly covered by the T-CREST project, three real-time operating systems (RTOS) have been ported to Patmos: (1) the Real-Time Executive for Multiprocessor Systems (RTEMS), (2) the time-composable operating system (TiCOS) [7], and (3) the research prototype of MOSSCA [31]. RTEMS is an open source RTOS, popular in avionics.[3] T-CREST with RTEMS has been used for the T-CREST evaluated with an avionic use case [49]. TiCOS is based on the open-source RTOS POK [11]. TiCOS and POK implement a two-level partitioned scheduler and provide an API according to the ARINC653 standard. Ziccardi ported TiCOS to Patmos [72].[4] MOSSCA is an operating system for safety-critical applications on manycore processors. The port to

[3] The port of RTEMS is available at `https://github.com/t-crest/rtems`.

[4] TiCOS is available at `https://github.com/t-crest/ospat`.

Patmos [32] explores the multicore version of Patmos and the usage of the SPMs.

## 4 The Architecture of Patmos

Patmos is a 32-bit RISC-style microprocessor optimized for time-predictable execution of real-time applications. To provide high performance for single-threaded code, we chose a two-way parallel architecture. To save hardware resources, Patmos is configurable as a two-way or single-way pipeline. For multi-threaded code, we provide a chip-multiprocessor system with TDM arbitration of the access to shared main memory.

Patmos is a statically scheduled, dual-issue RISC microprocessor. The processor supports bundles that are 32 or 64 bits wide. All instruction delays are explicitly visible at the ISA level. The compiler needs to respect the exposed delays from the pipeline to generate correct code. Knowing all pipeline delays and the conditions under which they occur simplifies the processor model required for WCET analysis and helps to improve accuracy. Furthermore, no instruction timing depends on either an operand value (e.g., a variable latency multiplication) or the execution of an earlier instruction, as even all cache misses are handled in the same pipeline stage. All instructions, except missing in the cache, have a constant execution time. These properties greatly simplify the computation of the WCET of basic blocks as there is no need to model the pipeline and instruction dependencies.

The modeling of memory hierarchies with multiple levels of caches is critical for practical WCET analysis. Patmos simplifies this task by offering caches specifically designed for WCET analysis. Accesses to different data areas are very different with respect to WCET analysis. Static data, constants, and stack-allocated data can easily be tracked by static program analysis. In contrast, heap-allocated data, to be analyzable, demands different caching techniques [27]. Therefore, Patmos contains two data caches, one for stack cache and one for other data. As the data cache will then cache unpredictable accesses to heap allocated data and to static data, we support cache bypassing load and store instructions for unpredictable data accesses. The WCET analysis tool aiT delivers the information on unpredictable data accesses back to the compiler. On a second compile run this information is used to generate cache bypassing load and store instructions. Furthermore, these instructions can also be used manually by the programmer.

### 4.1 Fully Predicated Instruction Set

The instruction set for Patmos is a RISC-style load/store instruction set that takes at most three register operands. However, in contrast to common RISC architectures, all instructions are fully predicated. While control-flow instructions and instructions that access memory can be executed only in

the first pipeline, arithmetic and logic instructions can be executed in both pipelines.

The first instruction of an instruction bundle contains a bit to encode the length of the bundle (32 or 64 bits). Register addresses are at fixed positions to allow reading the register file parallel to instruction decoding. The main pressure on the instruction coding comes from the size of constant fields and branch offsets. However, as we support fetching of up to two 32-bits words for the dual-issue pipeline, we use this feature to support ALU operations with 32-bit constants. The constant is encoded in the second instruction slot. Doing so enables the loading of 32-bit constants in a single cycle. Furthermore, most ALU instructions can be performed with a 12-bit constant operand, saving code space and leaving the second instruction slot free for other instructions. Patmos implements conditional and unconditional branches with a 22-bit offset. A register-indirect call instruction supports function calls to a 32-bit address.

To reduce the number of conditional branches and to support the single-path programming paradigm [46, 47], Patmos supports fully predicated instructions. Predicates are set with compare instructions, which can also be predicated. Patmos has 8 predicate registers.

Access to the different data areas (e.g., stack data) is explicitly encoded with the load and store instructions. This feature helps the WCET analysis to distinguish between the different data caches. Furthermore, which cache will be accessed can be detected earlier in the pipeline.

### 4.2 Dual-Issue Pipeline

Patmos contains 5 pipeline stages: (1) instruction fetch, (2) decode and register read, (3) execute, (4) memory access, and (5) register write-back. Figure 2 shows an overview of the Patmos pipeline.

The register file with 32 registers is shared between the two pipelines. Full forwarding between the two pipelines is supported. The basic features are like a standard RISC pipeline.

Figure 2 shows an overview of the Patmos pipeline. To simplify the diagram, forwarding and external memory access data paths are omitted. We can identify the program counter PC which delivers the fetch address for the method cache (M$). The M$ delivers the instruction and is the fetch stage. The instruction word(s) are stored in the instruction register (IR) for decoding in Dec. Furthermore, the register address fields from the instruction are input to the register file (RF). The RF is read in parallel with the instruction decoding. Instruction decoding and RF reading are the decode stage. The values of the register file and the instruction move to the next stage, the execution stage. In the execution stage, up to two ALU operations are computed or a memory address is computed. The next stage is the memory stage where a load or store operation is performed. The memory stage contains the stack cache (S$), the data cache (D$), and the scratchpad memory (SP). In the final stage, the write back stage, a computed result or a loaded word is written back into the RF.

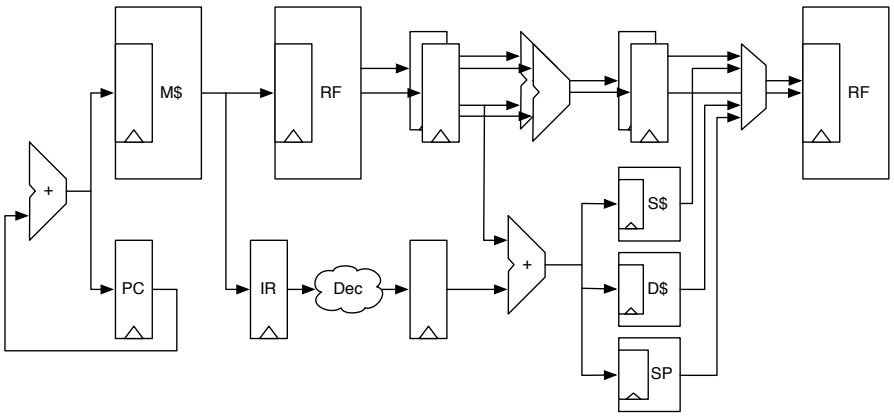


**Fig. 2** Dual issue pipeline of Patmos with fetch, decode, execute, memory, and write-back stages.

Due to the dual-issue pipeline, the register file needs four read ports and two write ports. Such a memory is not available in an FPGA. In the current design, we implement the register file with FPGA registers and use multiplexers for the read ports. If Patmos is configured with a single-issue pipeline, on-chip memory is used for the register file.

As Patmos provides full forwarding from both pipelines, this forwarding network consumes a lot of resources. If the full power of dual issue is not needed, Patmos can be configured as a single-issue pipeline.

## 4.3 Local Memories

Patmos contains three caches (method, data, and stack cache) and two scratchpad memories (SPM) for data and instructions. All caches are configurable in the size. To distinguish between the different caches, Patmos implements typed load and store instructions. The type information is assigned by the compiler (for the stack cache) or by the programmer (for the data SPM).

### *4.3.1 Boot ROM and Scratchpad Memories*

Patmos contains on-chip ROMs for instructions and data and a small SPM dedicated to bootstrapping, such that small applications and test cases do not need to access external memory. For larger applications, the boot ROM contains a boot loader that loads the application from a non-volatile memory (or, during development, from a serial line) into the main memory.

Patmos also contains (optional) SPMs for instructions and data. When code and/or data caching is under program control, these SPMs can be used in addition to caches or instead of caches. The usage of SPM is under programmer control. The multicore research operating system MOSSCA explored the usage of SPMs under operating system control [32].

#### *4.3.2 Method Cache*

Patmos contains a method cache that stores entire functions. The term "method cache" applies because this form of caching was originally introduced for a Java processor [51]. Caching entire functions means that functions may be loaded on a call or on a return. All other instructions are guaranteed cache hits. Our assumption is that those few possible miss points allow for an easier and more precise WCET analysis. Further details of the method cache for Patmos and its analysis appear in [10]. An average-case comparison between a method cache and a direct mapped instruction cache shows that the winner depends on the external memory properties [51]. With shorter latency and lower bandwidth, a standard cache performs better. With longer latencies and higher burst bandwidth, as found in modern SDRAM memories, a method cache performs better as it benefits from the higher bandwidth.

The assumption of a method cache is that the cache is larger than all individual functions in a program. However, this assumption cannot be guaranteed at the source code level. Furthermore, an optimizing compiler will inline functions to avoid the call and return overhead, leading to even bigger functions. To mitigate this issue, we have implemented a function splitting pass in the compiler that splits overly large functions into smaller (sub)functions that fit into the cache.

#### *4.3.3 Stack Cache*

For cache hit and miss classification, the address for the load or store instruction needs to be known. Some addresses, e.g., access to static data and stack access, are relatively easy to predict statically [29]. Addresses of heap-allocated data are known only at runtime and are therefore not statically predictable. Moreover, access to an unknown address destroys abstract information for one way in all sets of a cache in the cache analysis. Therefore, we split the cache into two caches: one for stack allocated data [1] and one for other data. Load and stores for the different caches are different instructions.

However, it is also possible to dynamically allocate data on the stack and furthermore pointers to stack allocated data can leak out of the function. In both cases we use a second stack, called the shadow stack, and those data are allocated on the shadow stack, which is cached by the standard data cache.

The stack cache [1] provides a window into the main memory address space. To manipulate the stack cache, Patmos has three instructions: (1) *reserve* reserves space in the stack cache, (2) *free* frees space on the stack cache, and (3) *ensure* enforces data to be in the stack cache. Only the *reserve* and *ensure* instructions may trigger a possible exchange with the main memory (spill and fill). All load and store instructions into the stack area are guaranteed hits. Stores into the stack area access only local memory; the write-back to main memory occurs when a *reserve* instruction causes a spill. Which stack manipulation instructions may lead to memory transactions can be statically

predicted [29]. As stacks are usually shallow, even a small stack cache provides good hit rates.

#### *4.3.4 Data Cache*

The current implementation of the data cache is a direct-mapped cache with write-through and no allocation on a write. Write-through was chosen, because WCET analysis tools do not track the state of a dirty bit in a write-back cache and therefore assume that a cache miss also needs a write back of this cache line. The design decision to use a write-through policy is yet another example of how WCET analyzability influences the hardware design for a time-predictable processor. For statically unknown load and store addresses, Patmos has load and store instructions that bypass all caches.

To mitigate the performance effects of the write-through policy, we implemented a small buffer that combines writes into bursts. Furthermore, the Patmos compiler includes a pass that replaces unpredictable accesses to the data cache with bypass load and store instructions.

Access to global (static) data should be easy predictable and a specialized cache for static data would be beneficial. In Java bytecode access to static data is performed with its own instructions, and we have implemented such a cache for static data in the Java processor JOP [52]. However, to distinguish between static and heap allocated data is not trivial in C. We consider providing compiler support to classify load and store instructions to static data as future work.

#### *4.3.5 Miss Detection and Pipeline Stalling*

The cache configuration of Patmos is unique with respect to miss detection: for all three caches, misses are detected (and the pipeline stalled) in the memory stage. This is normal for a data cache, but a standard instruction cache misses in the fetch stage. However, the method cache performs miss detection just on call and return. Therefore, these instructions can also stall in the memory stage.

The consequences of a single stalling pipeline stage are twofold: (1) The hardware implementation of stalling is simplified, and (2) cache analysis becomes simpler. As instructions that access memory are allowed only in the first pipeline, only one such instruction can be in the memory stage at one time. Therefore, no two instructions can trigger a cache miss in the same clock cycle for two caches. This feature contributes to a timing-composable architecture. Different caches can be analyzed independently and the results merged.

#### *4.3.6 Interrupts and Exceptions*

Patmos supports interrupts and exceptions with an exception unit that is mapped into the IO space. The general principle of operation is that the exception unit requests the execution of an exception from the pipeline, and the

pipeline acknowledges when it starts the execution of the respective exception handler.

The exception unit supports 32 exception vector entries, where exceptions 0 and 1 are reserved for the "illegal operation" and "illegal memory access" faults. While exceptions 2 to 15 can be used freely (e.g., by the operating system), exceptions 16 to 32 are attached to external interrupts, triggered by I/O devices. The default configuration of Patmos contains a programmable timer interrupt to support a real-time operating system, such as TiCOS ported to Patmos [72].

Instructions that stall the pipeline (loads, stores, calls, etc.) delay the triggering of interrupts until the pipeline resumes execution. Therefore, method cache fills or stack spills cannot be interrupted. Control-flow instructions delay the triggering of interrupts such that interrupts are never triggered inside a delay slot or while executing instructions speculatively. Multiplications delay the triggering of interrupts such that no multiplications are "in flight" when an interrupt handler is entered.

The return information for an exception, which is the base address of the method cache and the offset within the method cache (like a program counter), are stored in two special registers. No other hardware state needs to be stored by the processor on an interrupt. The interrupt handler must store and restore registers that it needs.

### 4.4 Multicore Architecture

Several Patmos processors share the main memory via a memory arbiter. Three versions of the arbiter are available: the Bluetree memory tree [19], a simple pseudo round-robin memory arbiter, and a distributed TDM arbiter [59]. The round-robin arbiter is a *pseudo* round-robin arbiter, as it does not perform the arbitration decision in a single clock cycle. The combinational decision in a single clock cycle does not scale to many cores. Instead, the arbiter sequentially "polls" each core for access. Therefore, for each core that does not need access to main memory, one clock cycle is "wasted." However, with enough cores the memory bandwidth is fully utilized, and all cores are basically memory bounded.

For external memory, we have implemented two different memory controllers for synchronous and asynchronous SRAMs and a time-predictable SDRAM controller [33].

## 5 Compiling and Worst-Case Execution Time Analysis for Patmos

Exposing the micro-architecture at the ISA level and requiring the application to manage local memories leads to a small and fast hardware design. It also simplifies the WCET analysis, because changes to the hardware state can be explicitly observed at the ISA level. The application code, and thus the

compiler, must ensure the efficient use of the available resources. While the compiler has a static view of the executed application, it also has a more high-level view than the processor. Together with less stringent resource and runtime requirements (as compilation is done at design time), the compiler can thus use more powerful optimizations.

The compiler supports the WCET analysis by supplying meta-information (about the program) that is available in the compiler but lost in the final binary to the analysis. Examples of such meta-information are indirect branch targets or possible addresses accessed by memory instructions. The compiler also uses feedback from the WCET analysis for optimization. For example, memory accesses—for which the value analysis of the WCET analyzer cannot determine the accessed address—are replaced by accesses directly to the external memory, so that such accesses do not introduce imprecision in the cache analysis.

### 5.1 Dual-Issue Support

The dual-issue architecture of Patmos requires the compiler to schedule instructions without hazards and to allocate instructions to the second pipeline. In our compiler, we use a standard bottom-up instruction scheduler to perform this task. Scheduling is performed after the insertion of register spill code and function prologues and epilogues. To break false dependencies between instructions, the compiler can use register renaming during scheduling. Depending on the number of instructions available for filling control flow delay slots, the compiler issues either delayed or non-delayed control flow instruction variants.

Supporting both delayed and non-delayed control flow instruction variants provides performance benefits in exchange for very little hardware cost. On the one hand, a fully filled delay slot provides better performance than a branch predictor can. On the other hand, even trivial "predict-not-taken" non-delayed branching provides better performance than executing only No-Ops. Our non-delayed branch instruction predicts non-taken, the simple version of just continuing instruction fetching after the branch. Combining the two variants provides better performance than either variant can alone. Compared to supporting only delayed branches, our approach also reduces the code size by suppressing No-Ops. The reduced code size, in turn, can reduce the method cache misses, thereby improving performance.

The compiler uses the fully predicated ISA to perform "if-conversion," i.e., the compiler converts small conditionally executed basic blocks into predicated straight-line code to avoid the overhead of branches. Our compiler can also eliminate all input-data dependent branches to generate single-path code.

### 5.2 Stack and Method Cache Support

The stack cache provides a fast way of storing temporary data without necessarily writing the data back to global memory. The compiler automatically

uses the stack cache for spilling registers and for stack-allocated data. Stack cache management instructions are inserted in the prologue and epilogue of functions, as well as after all call sites, to reserve and free stack frames in the stack cache [1]. However, pointers to data allocated to the stack cache cannot be passed to callees. Therefore, the compiler maintains a separate shadow stack in global memory. The shadow stack is used for dynamically allocated stack data and for data that can escape the function in which it is defined.

The method cache requires the compiler to split large functions into smaller sub-functions that fit into the cache. The compiler contains a function splitter pass that partitions functions into sub-functions of a predefined size. This feature enables the processor to execute arbitrarily large functions. Furthermore, when large functions are split into smaller sub-functions, the cache cost of larger functions, where not all code is executed, is reduced.

### 5.3 Single-Path Code Generation

The execution time of a given piece of code is determined by two factors: the sequence of actions along an execution path and the durations of these actions. Sources of uncertainty in execution time stem from both the software and the underlying hardware. Typically, different program inputs result in different execution paths. The duration of the instructions on these paths depends on the hardware state that has built up in the execution history.

The single-path approach removes the uncertainty on the software side. The key idea is to eliminate all input-data dependent control flow alternatives, and to construct a singleton execution path that is taken independent of the program inputs [46, 47]. The transformation is based on if-conversion [3], which turns input-data dependent control-flow alternatives into a straight-line sequence of predicated instructions.

The Patmos compiler can generate single-path code in an automated way [44]. It implements the single-path code transformation in the backend and operates on the control flow graph representation rather than on the source code. A guard predicate is computed for each basic block based on control dependences. The instructions of the basic blocks are then predicated with the corresponding guard. The guards are defined at the branch conditions of the original graph, and these defining instructions are predicated themselves. The basic blocks are put into topological order, and the branch instructions are removed.

This procedure is performed individually for each loop scope in the graph. Therefore, the forward control-flow graph of the loop (the graph without back edges) is considered. For loops with an input-data dependent iteration count, a new loop counter is introduced. This counter forces the code to be executed as many times as given by the local loop bound. The guards are defined in such a way that they are false for these excess iterations. The composition of the complete control-flow graph from the single-path loops is performed in a recursive manner: Inner loops are treated like single basic blocks and

assigned guards. The sketched transformation is general in that it can handle unstructured code (e.g., by `break` and `goto` statements) if the control flow graph is reducible [45]. The single-path transformation is applied on the level of functions. By specifying an entry function (a *single-path root*), the selected function itself and copies of all functions below in the call-graph are converted to single-path code. This limits the single-path transformation to programs without direct or indirect recursion. Note that recursion is discouraged or even prohibited in most coding standards for safety-critical software (e.g., MISRA C, DO178-B, ISO 26262-6).

The Patmos processor presents itself as a suitable target platform for single-path code, not only because of its predictable, fully predicated pipeline and the instructions for predicate register manipulation, but also because the timing-composable architecture allows for easy analysis and enables the generation of code with stable timing behavior.

Special instructions allow reading and writing a single bit within a 32-bit register. Furthermore, predicate live ranges coincide with the nesting depth of the control-flow graph. During successive loop iterations, the predicates of the code outside the loop do not need to be accessed. Therefore, the compiler allocates predicate registers within loop scopes. When a loop is entered, the whole set of live predicate registers is stored such that the registers are available for the predicates inside the loop. When the loop is left, the set is restored. Patmos supports storing and restoring of the complete predicate register file at once by making it accessible as a special register.

Obtaining the WCET for single-path code on Patmos is straightforward. The latency of an instruction (except for pipeline stalls at memory accesses) is independent of the value of its operands, its predicate operand. To obtain the WCET, the memory access-related worst-case costs must be added to the number of instructions on the singleton execution path.

In single-path code, function calls and the corresponding stack cache allocations are performed unconditionally. Thus, both the sequence of (sub-)functions loaded into the method cache and the sequence of stack cache control operations is invariant. The related worst-case latencies can be obtained by simulation with a simple hardware model.

If a data cache is used, the memory access costs of a conventional data cache analysis performed on the original (not single-path converted) control-flow graph can be added. A more suitable alternative to the data cache would be the usage of the data SPM, where single-cycle access costs are given once the data is loaded to the local SPM, and the costs for memory write-through are avoided. In this case, the costs for the (explicit) transfer of the data between the local SPM and the global memory must be added.

The single-path approach is an orthogonal approach to code predictability. The knowledge of one execution of a single-path task accurately predicts every execution of the task. Tasks generated as single-path code exhibit a predictable and stable timing behavior with little or no execution time jitter. This is a stronger property than WCET predictability, and certainly, it comes at a higher cost. Depending on the program structure, the overall length of the singleton

execution path may be increased compared to any path through the original, conventionally compiled program. The programmer specifies the tasks to be generated as single-path code either by function attributes in the source code or by passing the names of the functions to the compiler. Because of the cost, they will only apply single-path code generation for tasks for which such a predictable, stable timing behavior is desired.

### 5.4 Worst-Case Execution Time Analysis

Static WCET analysis has been an integral part of the Patmos toolchain since its early developmental stages. In addition to precise WCET bounds, an important goal was to reuse existing platform-independent program analyses, to benefit from advances in the rapidly evolving static analysis field. To achieve this goal, all information necessary for WCET analysis should be provided by the compiler and by analysis tools that operate on the platform-independent bitcode representation of LLVM. In this approach, a major challenge is the transformation and combination of compiler and analysis information. The `platin` tool that was developed for Patmos [48] performs these transformations and combinations.

The `platin` tool supports the transformation of platform-independent flow information to machine code, using an approach that ensures sound results [26]. Furthermore, it prepares relevant analysis information for external binary-level WCET analysis tools, e.g., the well-known industrial tool aiT [22]. The `platin` tool communicates with the compiler using PML (Platin Metainfo Language) files, which contain all information (about the analyzed program) relevant for WCET analysis. A distinguished feature of PML is that it allows to store information about both the platform-independent intermediate representation and the platform-dependent machine code.

Figure 3 illustrates the integration of the compiler, WCET analysis, and external tools into our framework. The application code is translated into bitcode and linked at this intermediate level by the compiler (patmos-clang). At this point the compiler has a complete view of the application. The compiler (patmos-llc) and linker (patmos-gold) produce the executable binary file and additional produces a PML, which contains structural information and flow information derived from bitcode analyses and program annotations. With `platin` this information can be extended. For example, our toolchain supports the external tool SWEET [36] to obtain more precise flow-information. To this end, the bitcode is translated to ALF, the input format of SWEET. The tool generates target-independent flow facts that are added to the PML file. `platin` translates the available flow information to the machine code representation. A set of exporters (e.g., platin-aiT and platin-otawa) exists, which output the information to a format suitable as input to external WCET analysis tools, like the aiT [22] or OTAWA [8].

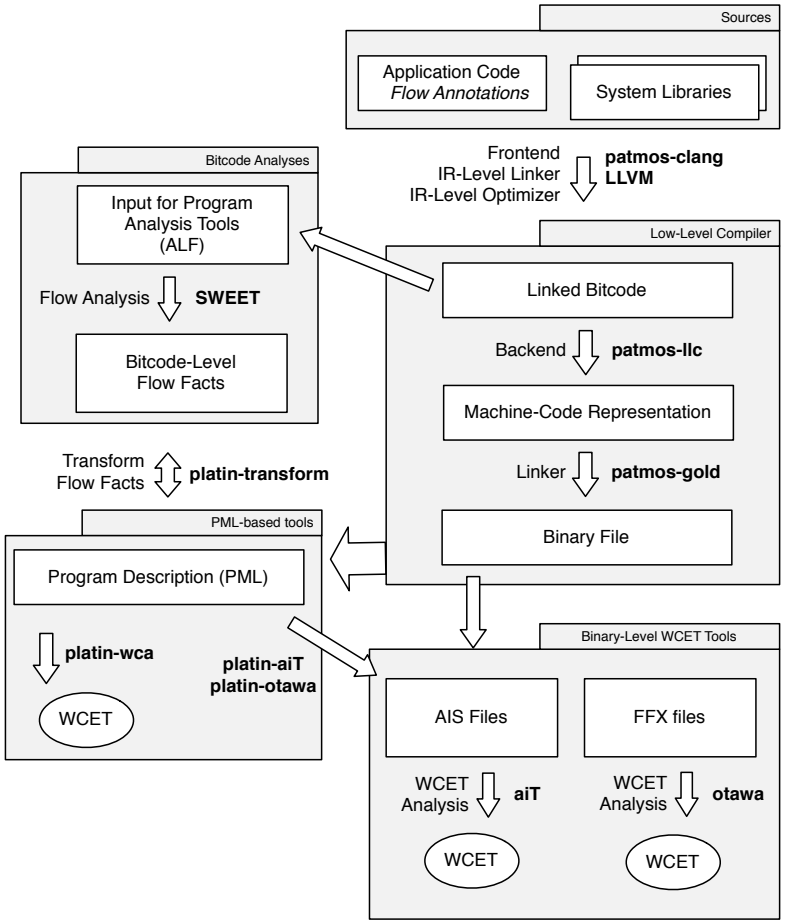


**Fig. 3** The T-CREST approach to compilation and integration with WCET analysis.

Additionally, `platin` provides a WCET analysis tool on its own (platin-wca), thereby taking advantage of three features that are characteristic for the Patmos architecture:

First, our static analysis operates (almost) exclusively on information that was provided by the compiler and platform-independent analyses. We therefore avoid duplicating efforts of the compiler and do not need to model the semantics of machine code in detail.

Second, the Patmos design ensures that the timing of hardware components can be analyzed independently. This feature allows us to decouple different cache analyses and the pipeline timing analysis, and to use global cache analyses (e.g., persistence analyses) that avoid costly virtual loop-peeling.

Third, the Patmos design prevents the need for excessive context sensitivity for hardware timing analysis. This feature avoids scalability problems and

allows modular analyses. For example, stack accesses cached by a conventional data cache do not pose a problem if the value of the stack pointer is known, but they are unpredictable if full virtual inlining is intractable. In contrast, our analysis of the stack cache [29] does not require full context sensitivity.

The `platin` tool, including the WCET analysis, is easy to adapt for different research experiments, and is deployed as part of the open-source Patmos compiler.

## 6 Implementation

A software simulator of the architecture is the first, important step for exploring ideas and for serving as a reference design for the compiler and the hardware design. Therefore, we provide a software simulator and a hardware implementation of Patmos.

### 6.1 Simulator

At the start of the development of Patmos, we developed a cycle-accurate software simulator. This simulator serves as the reference for the hardware implementation of Patmos, for the development of the compiler, and the porting of real-time operating systems. Furthermore, the simulator provides variants of caches and memory controller models, and can thus be used for design space exploration of caches.

### 6.2 Hardware Implementation

We use Chisel [6] for the implementation and simulation of the core design. Chisel, developed at UC Berkeley, is a hardware-construction language, embedded in the programming language Scala. Consequently, Chisel allows the programmer to design efficient hardware components in a high-level language. Scala, and therefore Chisel, are object-oriented and functional languages, enabling hardware design in an object-oriented way.

The Chisel back-end can generate both Verilog and C++ code. While Verilog is used to implement a design on an ASIC or FPGA, the C++ code implements a fast high-level simulation of the hardware and provides a test environment. We call the Chisel-generated C++ simulation the "emulator", to distinguish it from the software-based simulator.

We adapted the top-level class of the emulator and added a model of an external SRAM memory. Furthermore, an executable file can be loaded into the memory (or optionally into the scratchpad memory of the model) to start the execution. The emulator produces a precise model of the system behavior and also the possibility of easily inserting debug information for explicit testing.

Because the Patmos emulator is auto-generated from the hardware description, it lends itself to the high-level debugging and testing of the processor

implementation. In contrast to the simulator, the individual registers and signals of the hardware design are emulated. We use the emulator to verify the cycle-accurate behavior of the Patmos simulator.

The use of Chisel facilitates the configuration of the hardware implementation. Cache sizes, the number of pipelines (dual- or single-issue), and other features can be controlled through a single XML configuration file. Moreover, I/O devices can be added to the processor through this configuration file. Parsing the configuration is done in Chisel, making code generation steps or manual editing of the code unnecessary.

### 6.3 Co-Simulation

When building a complex hardware, such as the Patmos processor, testing, having good test coverage—and checking the outcome of the tests—is very important. The software simulator can serve as the gold reference for the hardware implementation of Patmos.

We execute test cases on both the simulator and the emulator and compare the execution traces of both executions on a cycle-by-cycle base. To compare the two simulations, we consider the most important state of a processor: the register file. Although a difference in another program visible state (program counter, predicate registers, main memory) might also be interesting, a difference there will at some point (i.e., some cycles later) show up in the register file. If not, the failure would not be visible during a normal program execution. As all loads and stores pass through the register file, separately monitoring the state of the main memory is unnecessary. A collection of assembler programs is co-simulated automatically every night.

### 6.4 Testing and Validation

Apart from a small set of test cases written in assembly, we use an extensive test suite with test cases written in C for testing and validation. This testing ensures that the compiler, the simulator, and the hardware consistently implement the ISA. The test suite includes the MiBench[5] and the Mälardalen [40] benchmarks. Furthermore, the test suite includes the `gcc.c-torture/execute` test cases from GCC's test suite, which covers a wide range of corner cases for compilation and execution. In total, the test suite contains more than 1000 test programs, yielding to more than 2000 individual test cases. As the test suite is executed every night, we will quickly notice regressions in the code base.

[5] `http://www.eecs.umich.edu/mibench/`

| Configuration | Resources (LC) | Memory (KB) | fmax (MHz) |
|---|---|---|---|
| Standard | 15,320 | 13.6 | 78.4 |
| Large caches | 16,395 | 38.5 | 80.1 |
| Single issue | 9,193 | 13.9 | 84.6 |
| Minimal | 7,602 | 5.7 | 81.7 |

**Table 1** Resource consumption and maximum clock frequency of different Patmos configurations

## 7 Evaluation

In addition to a processor's being time-predictable, it has two other important properties: (1) its size and (2) its performance. We present results of Patmos from an implementation in a low-cost FPGA.

### 7.1 Resource Consumption

Patmos is highly configurable with respect to the resource consumption. In this section, we show results for Patmos in four different configurations: (1) the standard configuration, (2) with large caches, (3) single issue, and (4) minimal.

All results are from synthesizing Patmos for an Altera Cyclone IV FPGA (EP4CE115F29C7) and with a memory interface to the 16-bit asynchronous SRAM on the Altera DE2-115 FPGA board. All configurations include several IO devices and an exception unit. The maximum clock frequency is reported for the slow timing model at 1200 mV core voltage and at 85 C core temperature. For synthesize we used Quartus Prime Lite Edition version 15.1.0. Every synthesize option is set to the default values.

Table 1 compares the results for the four configurations of Patmos. The table shows hardware resource consumption in logic cells (LC) and on-chip memory. An LC of the Cyclone IV FPGA contains a 4-bit lookup table and a register. The last column shows the maximum frequency. The resource consumption shown is for a full system with several caches and SPMs, IO devices, and a memory controller, not just the processor pipeline.

The standard configuration of Patmos is dual-issue execution, a method cache of 4 KB with maximum 16 methods, a direct-mapped data cache of 2 KB, a 2 KB stack cache, an instruction SPM of 2 KB, a boot ROM of 1 KB, a data SPM of 2 KB, and a memory controller for the external asynchronous SRAM.

The large configuration is the configuration used for the average case benchmarking with CoreMark. The caches are increased to a 16 KB method cache, a 8 KB data stack, and a 8 KB stack cache. Compared to the standard configuration of Patmos, we can see the increase in on-chip memory usage and about 1000 additional LCs, which are needed for the larger caches.

The third configuration of Patmos is the same as the standard configuration, but configured with a single-issue pipeline. We see that the single-issue version of Patmos reduces the footprint by 44%. The main savings are in the decode

| Component | Resources (LC) | Memory (KB) |
|---|---|---|
| Fetch | 716 | 3 |
| Decode | 4,681 | 0 |
| Execute | 4,409 | 0 |
| Memory | 589 | 0 |
| Data cache | 649 | 2.3 |
| Stack cache | 957 | 2 |
| Method cache | 1,998 | 4 |
| IO | 1,066 | 2 |
| Exception unit | 477 | 0.2 |
| SRAM controller | 495 | 0 |
| Total | 15,370 | 13.6 |

**Table 2** Resource consumption of Patmos components in the standard configuration

stage and the execution stage. The register file, which is included in the decode stage, can now be implemented in on-chip memory. Therefore, we can see a small increase in memory consumption (0.25 KB). With a single-issue configuration, the biggest resource saving comes from the simpler forwarding network. In the execution stage, only a single ALU is needed.

For the minimal configuration of Patmos, we reduced all caches to 1 KB and all SPMs to 0 KB, thereby reducing the amount of on-chip memory. In the minimal configuration, the method cache is also restricted to cache only two methods. As the tag memory for the method cache is implemented in LCs, the number of LCs is therefore also reduced. This configuration also contains only a single-issue pipeline.

In Table 1 we can see that the single issue-pipeline results in the highest clock frequency. At at first look, it seems counterintuitive that a configuration with larger caches can lead to a higher clock frequency. However, larger caches might lead to a better alignment of on-chip memory blocks resulting in a slightly higher frequency (about 2 %). Similar, the minimal configuration has a slightly lower clock frequency than the standard version. Maximum clock frequency in FPGAs is not directly correlated to the size of a design. Sometimes larger designs force a denser packing of logic elements, resulting in a higher clock frequency.

Table 2 shows the resource consumption of the individual components in the standard configuration in logic cells (LC) and on-chip memory (KB). We can see 4 of the 5 pipeline stages as dedicated components: Fetch, Decode, Execute, and Memory. Write-back is merely the write port of the register file and therefore not visible as a hardware component.

The fetch stage is relative small, containing just the program counter and some multiplexing supporting unaligned fetch of a 64-bit dual-issue instruction word. It also contains the 2 KB instruction SPM and the 1 KB ROM for the boot loader. The decode stage contains the register file. As this register file is built out of LCs for the dual-issue version of Patmos and we support full forwarding between the two pipelines, the resource consumption is high.

It is mainly dominated by the forwarding paths. Related to the dual-issue configuration is the size of the execution stage, as it contains the full forwarding from the memory and write back-stages of both pipelines to both execution stages. The memory stage contains just the addressing and multiplexing of various caches and input/output (IO) components mapped into memory.

The data and stack cache each contain a 2 KB memory; the data cache additional 0.3 KB for the tag memory. The stack cache has no tag memories as the content is determined by two pointers into the stack memory. Another component that contributes significantly to the resource consumption is the method cache. As the tag memory for 16 methods is fully associative, it is build out of dedicated registers.

The IO component includes interfaces to switches and LEDs, a timer, a UART, and the 2 KB data SPM. The exception unit supports interrupts and runtime exceptions. The SRAM controller interfaces to a 16-bit SRAM. However, the standard interface to Patmos is a 4-word burst interface according to the Open Core Protocol specification. Therefore, the memory controller also contains the translation between the burst interface and the memory interface.

Note, that the total number of LCs is less than the sum of LCs for all components. This can be explained by some components, e.g., the forwarding logic in the decode stage, uses more LUTs and less registers, but another component, e.g., the tag memory of the method cache, uses more registers. Therefore, LCs are shared between components.

### 7.2 Average-Case Performance

To evaluate the average-case performance of Patmos, we use the CoreMark benchmark. We selected CoreMark as this is a popular benchmark for embedded processors and because the web site of CoreMark contains many scores from different processors.[6] We compare the results to other FPGA-based processors. We compare Patmos to the Aeroflex Gaisler LEON3, Xilinx MicroBlaze, and Altera NIOS II processors.

Most of the available CoreMark scores for these processors were obtained with 16 KB instruction cache and 16 KB data cache. We use a comparable configuration of Patmos for this evaluation, with a 16 KB method cache that can hold 16 methods, an 8 KB data cache, and an 8 KB stack cache. The SPMs are disabled. We used the DE2-70 FPGA board with the Cyclone II FPGA for the average case performance measurement. With such a setup, the CoreMark benchmark fits into the caches, so that the benchmark evaluates the processor pipeline rather than the efficiency of the memory subsystem.

Table 3 shows CoreMark scores relative to the operating frequency, the absolute CoreMark score, the operation frequency, and the FPGA type used for Patmos and three other softcore processors. The results for Patmos, LEON3, NIOS II, and MicroBlaze were obtained on different FPGAs. Therefore, the

[6] CoreMark scores for LEON3, MicroBlaze, and NIOS II are from `http://www.eembc.org/coremark/index.php`, last accessed 29 November 2016.

| Processor | CoreMark/MHz | CoreMark | Frequency | FPGA |
|---|---|---|---|---|
| Patmos, dual-issue | 2.19 | 175 | 80 MHz | Cyclone II |
| Patmos, single-issue | 1.97 | 158 | 80 MHz | Cyclone II |
| LEON3 | 1.96 | 196 | 100 MHz | Spartan-6 |
| MicroBlaze | 1.75 | 175 | 100 MHz | Virtex 4 |
| MicroBlaze | 1.90 | 238 | 125 MHz | Virtex 5 |
| NIOS II | 1.29 | 64 | 50 MHz | Cyclone III |
| NIOS II | 1.49 | 119 | 80 MHz | - |
| NIOS II/f | 1.87 | 187 | 100 MHz | - |

**Table 3** CoreMark scores for Patmos, LEON3, MicroBlaze, and NIOS II

operation frequency and the absolute CoreMark scores are incomparable and we also show the CoreMark/MHz measure to evaluate the efficiency of the instruction set and the compiler. However, for the last two results no information on the used FPGA is available.

We can see that Patmos performs in the same range as the other processors. Just comparing the relative performance, Patmos can beat all the other processors. We conclude that the performance of the Patmos pipeline is in the same range as comparable processors that are not optimized for time predictability. This is expected as all processors have a similar pipeline structure as a RISC processor. Therefore, we conclude that with our pipeline architecture we did not reduce average case performance by optimizing for the WCET.

The speedup of the dual-issue version of Patmos relative the single-issue is merely 10.8%. As the benchmark fits into the cache, this benchmark is not memory bound. This means that the second pipeline is underutilized. The result indicates that there is room for improvements in the compiler, especially in the instruction scheduler. Only a simple instruction scheduler for the dual-issue feature of Patmos has been developed. Within this project most of the compiler work has been focused on integration with worst-case execution time analysis [48,26], single-path code generation [44], and function splitting [24]. Future work on the compiler is necessary for using the second pipeline more efficiently, e.g., by software pipelining [34] and trace [18] or superblock [28] scheduling.

The code size for the single-issue executable is 355 KB and the dual-issue executable is 359 KB. This executable also includes compiler support library code (e.g., software division) and standard library code (e.g., printf()). We see only a slight increase in code size in the range of 1 % for the dual-issue executable. For comparison, CoreMark compiled under Intel 64-bit Linux is 263 KB. Besides possible differences in the libraries and different compilers (LLVM versus gcc) it is expected that the executable size for the Intel processor is smaller than the executable size for a RISC style instruction set.

To make Patmos easily accessible, we support in the standard configuration the relative cheap FPGA boards DE2-70 and DE2-115. These board contain FPGAs from the low-cost Cyclone series form Intel/Altera. To get an idea on the possible performance of Patmos in different FPGAs from Intel/Altera we

| FPGA | Frequency |
|---|---|
| Cyclone IV | 94.2 MHz |
| Cyclone V | 110.6 MHz |
| Stratix IV | 167.8 MHz |
| Stratix V | 206.9 MHz |

**Table 4** Maximum clock frequency of Patmos in different FPGAs.

synthesized the core for different FPGA families. To measure the limits of the maximum clock frequency we configured the PLL to produce a 200 MHz clock, selected the fastest speed grade of the FPGA, set optimize for performance, and let the tool select the device. Setting the PLL to 200 MHz tells the synthesize tool that this is the minimum requested frequency. For three out of the four FPGA this is over constraining the design.

In Table 4 we report the maximum frequency using the slow timing model at 85 C. We can see that by optimizing for performance and using the fastest speed grade it is possible to increase the clock frequency on the Cyclone IV FPGA from about 80 MHz to 94 MHz. Using the newest supported version[7] of the high-performance FPGA family from Intel/Altera, the Stratix V we can clock Patmos at about 207 MHz, easily achieving the minimum clock frequency constraint of 200 MHz. Ignoring the memory subsystem, this would increase the CoreMark for Patmos by a factor of 2.5.

### 7.3 Worst-Case Execution Time

Patmos is designed as an easy target for worst-case execution time analysis. For our evaluation, we used the static WCET analysis tool of the `platin` toolkit.

For the WCET evaluation, we used the standard configuration of Patmos with dual-issue execution, a method cache of 4 KB with maximum 16 methods, a direct-mapped data cache of 2 KB, a 2 KB stack cache, an instruction SPM of 2 KB, a boot ROM of 1 KB, a data SPM of 2 KB, and a memory controller for the external asynchronous SRAM of the DE2-115 board with 21 clock cycle latency for a 4-word burst transfer. The information emitted by the compiler already allows for a fully automatic WCET analysis, i.e., without the need for manually adding loop bounds. We use a scope-based method cache analysis [25] and an analysis of the stack cache [29], which are implemented in the WCET tool of `platin`. For the data cache, we use a conservative analysis (assume always-miss).

We use PapaBench for the evaluation of the WCET analysis, as this is a very popular benchmark in the WCET analysis community. Furthermore, PapaBench is derived from a real application. Table 5 summarizes the results of the WCET analysis of the PapaBench WCET benchmark. PapaBench consists of two application binaries, `fly_by_wire` (abbr. `fbw`) and `autopilot` (abbr. `auto`).

[7] Stratix 10 is not (yet) supported in the latest Quartus version.

| Benchmark/Task | WCET Bound | WCET Tests | MOET Tests | Number of Calls |
|---|---|---|---|---|
| papa_auto/altitude_control | 12883 | 4781 | 4256 | 25 |
| papa_auto/climb_control | 58889 | 31851 | 21229 | 200 |
| papa_auto/link_fbw_send | 670 | 628 | 544 | 3 |
| papa_auto/navigation | 522606 | 137059 | 69082 | 25 |
| papa_auto/radio_control | 155825 | 159 | 159 | 1 |
| papa_auto/receive_gps | 193679 | 621 | 432 | 1 |
| papa_auto/reporting | 79251 | 9493 | 6511 | 1 |
| papa_auto/stabilisation | 49949 | 22954 | 16822 | 1 |
| papa_fbw/check_failsafe | 72289 | 8165 | 7913 | 12 |
| papa_fbw/check_mega128 | 72612 | 8445 | 8151 | 16 |
| papa_fbw/send_data | 30768 | 8737 | 7183 | 16 |
| papa_fbw/servo_transmit | 7944 | 7857 | 2796 | 2 |
| papa_fbw/test_ppm | 133515 | 42155 | 30365 | 16 |

**Table 5** PapaBench WCET analysis results with conservative data cache analysis

Each binary schedule several real-time tasks that need to be analyzed. The column *WCET Bound* shows the statically computed WCET bound for the corresponding task, without adding any additional flow information other than provided by the compiler. To allow us to obtain some numbers on the actual (average) execution time, we created some tests where we execute the tasks with different values for the global variables, which determine the runtime behavior of the tasks. The maximum observed execution time (MOET) for the tests is shown in the fourth column *MOET Tests*. Finally, to compare static analysis and measurements, we also computed a WCET bound for the execution time of a task's test run. We did not explore which state of the variables will cause the worst-case path to be executed. Instead, we obtain the WCET for the test runs by limiting the execution frequency of each basic block to the maximum execution frequency observed during testing, which are recorded by the trace-analysis tool of `platin`. This reduces the over-approximation caused by the uncertainty of the program flow. Therefore, the column *WCET Tests* provides a sound upper bound for the maximum observed execution time during testing. In addition, column *Number of Calls* of Table 5 shows how often each task was called during testing.

The results in Table 5 show that the gap between *WCET Tests* and *MOET Tests* is in the range from no gap to +180%. To gain a deeper understanding of the sources of the over-approximation, we also evaluated the benchmark with a setup using an ideal data cache and otherwise identical configuration. This allows to mask the pessimism of the conservative data cache analysis. Table 6 shows the results of this evaluation. In this configuration, for a third of the tasks the numbers WCET Tests and MOET Tests match, and for more than half of the tasks the over-approximation of the upper bound is below 5%. Tasks with a higher gap between the test-specific WCET bound and the observed worst case include the *navigation* and *climb_control* tasks. For these tasks, information about mutually exclusive program paths would benefit the analysis but is not included in the flow facts recorded from the trace analysis.

| Benchmark/Task | WCET Bound | WCET Tests | MOET Tests |
|---|---|---|---|
| papa_auto/altitude_control | 9778 | 3857 | 3668 |
| papa_auto/climb_control | 44243 | 23251 | 16568 |
| papa_auto/link_fbw_send | 40 | 40 | 40 |
| papa_auto/navigation | 380678 | 116143 | 62068 |
| papa_auto/radio_control | 105770 | 117 | 117 |
| papa_auto/receive_gps | 135097 | 495 | 306 |
| papa_auto/reporting | 25460 | 4537 | 4327 |
| papa_auto/stabilisation | 36594 | 17830 | 13840 |
| papa_fbw/check_failsafe | 50239 | 6191 | 6191 |
| papa_fbw/check_mega128 | 50352 | 6303 | 6303 |
| papa_fbw/send_data | 20142 | 4999 | 4789 |
| papa_fbw/servo_transmit | 1308 | 1305 | 822 |
| papa_fbw/test_ppm | 97077 | 23579 | 20243 |

**Table 6** PapaBench WCET analysis results, assuming an ideal data cache

The effect is amplified as for the computations in the different paths floating-point arithmetic is performed, which is done in software for Patmos. Tasks *stabilization* and *receive_gps* were called only once each. The gap in this case indicates the over-approximation caused by the method cache and stack cache analyses.

It is worth emphasizing that the flow-information emitted by the compiler is determined independently from the target. The information from high-level analyses on bitcode (which were already present in LLVM) is transformed by `platin` to the machine-code representation. In this benchmark, they are sufficient to perform WCET-analysis without additional manual annotations. Also, the results with the data cache are expected to become better when more precise data cache analyses are added to the `platin` toolkit. We plan to add a standard data cache analysis, such as [17], to `platin` in future work.

### 7.4 Single-Path Code

Patmos is a suitable target for single-path code, due to its fully predicated instruction set, its predictable pipeline, and its controllable memory architecture. In our evaluation for the code produced by the single-path code generator, we assume that all global data is available locally. Stack allocated data is still cached in the stack cache. In this regard, the setup is identical to the one used in Section 7.3 with the ideal data cache. We compare the execution time behavior of single-path code to the execution time behavior of the conventional code in the worst case and the execution time jitter.

Table 7 shows the comparison between the conventionally compiled code and the single-path code. The statically computed worst case bound for the tasks compiled conventionally is shown in column *WCET Bound*. The range of their observed execution times is shown in column *Exec-Time (Conv.)*. The column *Single-Path* shows the execution time for the single-path code. The

| Benchmark/Task | WCET Bound (Conv.) | Exec-Time (Conv.) | Single-Path Exec-Time | Ratio |
|---|---|---|---|---|
| papa_auto/altitude_control | 9778 | [29,3668] | 9076 | 0.928 |
| papa_auto/climb_control | 44243 | [88,16568] | 67738 | 1.531 |
| papa_auto/link_fbw_send | 40 | [11,40] | 50 | 1.250 |
| papa_auto/navigation | 380678 | [4636,62068] | 2966730 | 7.793 |
| papa_auto/radio_control | 105770 | [117,117] | 137165 | 1.297 |
| papa_auto/receive_gps | 135097 | [306,306] | 186842 | 1.383 |
| papa_auto/reporting | 25460 | [4327,4327] | 26230 | 1.030 |
| papa_auto/stabilisation | 36594 | [13840,13840] | 45824 | 1.252 |
| papa_fbw/check_failsafe | 50239 | [37,6191] | 66254 | 1.319 |
| papa_fbw/check_mega128 | 50352 | [161,6303] | 66398 | 1.319 |
| papa_fbw/send_data | 20142 | [26,4789] | 19414 | 0.964 |
| papa_fbw/servo_transmit | 1308 | [822,822] | 1175 | 0.898 |
| papa_fbw/test_ppm | 97077 | [213,20243] | 112246 | 1.156 |

**Table 7** Single-path code evaluation

execution time is identical for every execution run, i.e., it constitutes the worst-case execution time. This is achieved by construction, as the execution follows only one path and the method cache and of the stack cache are put in a well-defined state by flushing them before the task's execution. The last column, *Ratio*, shows the ratio of the execution time of the single-path task to the worst-case execution time of the task compiled conventionally, i.e., *Single-Path Exec-Time/ WCET Bound (Conv.)*.

While in most of the cases the (worst-case) execution time of the single-path code is higher than the computed WCET bound for the conventional code, in some cases it is lower. This can be attributed to one of the following reasons. The basic blocks of single-path code are larger, such that the VLIW scheduler has more opportunity to bundle instructions and increase the ILP. Also, there is no overhead caused by over-approximation in the analysis for the single-path code. One of the main advantages of single-path code is that on predictable architectures like Patmos its execution time behavior is almost trivial to obtain. The most limiting drawback is the performance impact of single-path code generation, if applied blindly. For example, the `papa_auto/navigation` benchmark either calls a function `nav_home()` or `nav_update()`, depending on the current operating mode. The latter function implements a multi-level state machine. In the single-path variant, all actions of this state machine are serialized, resulting in an almost 7.8 times higher execution time than the statically computed bound for the conventional variant.

For this evaluation, we assume an ideal data cache to eliminate uncertainty in execution time behavior due to data accesses. In practice, the data SPM would be used for fast and time-invariant data access during the execution of the single-path task. At the time of writing, this can be achieved only by the programmer by explicitly moving data from and to the data SPM. Ideally, a compiler pass would allocate data to the data SPM automatically, but unfortunately this has not been implemented yet.

Automatic allocation of data in SPM is not trivial due to pointer aliasing. However, with some hardware support to translate main memory addresses to SPM addresses, this aliasing problem can be solved. The scratchpad memory management unit [70] introduces this mechanism as an enhancement to an SPM. This proposed solution does not require whole-program pointer analysis and makes load and store operation time-predictable.

## 8 Conclusion

The basis for time-predictable computing systems is a time-predictable processor. This paper presented such a time-predictable processor, the open-source design Patmos. To support real-time systems, all architectural features of Patmos are optimized for the worst-case execution time instead of the average-case execution time.

Patmos contains a statically scheduled dual-issue pipeline for good performance without the unpredictability of dynamic instruction issuing. Patmos contains a method cache, a stack cache, and a data cache, thereby providing caches that are easy to analyze.

We have compared the average case performance of Patmos with other RISC processors targeting an FPGA. Although those processors are optimized for average-case performance, Patmos provides similar performance for the CoreMark benchmark.

We have shown WCET analysis results with our open-source toolkit on the popular PapaBench benchmark, focusing on hardware predictability. The timing-compositional nature allows for separate, specialized analyses. We also evaluated the performance of automatic single-path code generation. Applied carefully, a predictable and stable code timing can be obtained at acceptable additional cost.

**Acknowledgements** We would like to thank Tommy Thorn for the ongoing discussions on computer architecture, processor design, and optimization for an FPGA implementation. We would like to thank Florian Brandner for discussions on the Patmos instruction set, the initial implementation of the software simulator of Patmos, and the initial port of LLVM for Patmos. We would like to thank Sahar Abbaspour for helping on a first VHDL version of the pipeline.

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). This work is part of the project "Hard Real-Time Embedded Multiprocessor Platform - RTEMP" and received partial funding from the Danish Research Council for Technology and Production Sciences under contract no. 12-127600.

## References

1. Abbaspour, S., Brandner, F., Schoeberl, M.: A time-predictable stack cache. In: Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems (2013)

2. Akesson, B., Goossens, K., Ringhofer, M.: Predator: a predictable sdram memory controller. In: CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pp. 251–256. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1289816.1289877
3. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: Proc. 10th ACM Symposium on Principles of Programming Languages, pp. 177–189 (1983)
4. Arnold, R., Mueller, F., Whalley, D., Harmon, M.: Bounding worst-case instruction cache performance. In: IEEE Real-Time Systems Symposium, pp. 172–181 (1994)
5. Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., Sebastian, M., Hanxleden, R.V., Wilhelm, R., Yi, W.: Building timing predictable embedded systems. ACM Transactions on Embedded Systems (2013)
6. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In: P. Groeneveld, D. Sciuto, S. Hassoun (eds.) The 49th Annual Design Automation Conference (DAC 2012), pp. 1216–1225. ACM, San Francisco, CA, USA (2012)
7. Baldovin, A., Mezzetti, E., Vardanega, T.: A time-composable operating system. In: T. Vardanega (ed.) 12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy, *OASICS*, vol. 23, pp. 69–80. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
8. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An Open Toolbox for Adaptive WCET Analysis, pp. 35–46. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-16256-5_6
9. Brandner, F., Hepp, S., Jordan, A.: Criticality: static profiling for real-time programs. Real-Time Systems pp. 1–34 (2013). DOI 10.1007/s11241-013-9196-y
10. Degasperi, P., Hepp, S., Puffitsch, W., Schoeberl, M.: A method cache for Patmos. In: Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014), pp. 100–108. IEEE, Reno, Nevada, USA (2014). DOI 10.1109/ISORC.2014.47
11. Delange, J., Lec, L.: POK, an ARINC653-compliant operating system released under the BSD license. In: 13th Real-Time Linux Workshop, vol. 10 (2011)
12. Delvai, M., Huber, W., Puschner, P., Steininger, A.: Processor support for temporal predictability – the SPEAR design example. In: Proceedings of the 15th Euromicro International Conference on Real-Time Systems (2003)
13. Edwards, S.A., Kim, S., Lee, E.A., Liu, I., Patel, H.D., Schoeberl, M.: A disruptive computer design idea: Architectures with repeatable timing. In: Proceedings of IEEE International Conference on Computer Design (ICCD 2009). IEEE, Lake Tahoe, CA (2009)
14. Edwards, S.A., Lee, E.A.: The case for the precision timed (PRET) machine. In: DAC '07: Proceedings of the 44th annual conference on Design automation, pp. 264–265. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1278480.1278545
15. Falk, H., Kleinsorge, J.C.: Optimal static WCET-aware scratchpad allocation of program code. In: DAC '09: Proceedings of the Conference on Design Automation, pp. 732–737 (2009)
16. Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems pp. 1–50 (2010)
17. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Systems **17**(2-3), 131–181 (1999)
18. Fisher, J.A.: Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers **C-30**(7), 478–490 (1981). DOI 10.1109/TC.1981.1675827
19. Garside, J., Audsley, N.C.: Investigating shared memory tree prefetching within multimedia noc architectures. In: Memory Architecture and Organisation Workshop (2013)
20. Gomony, M.D., Akesson, B., Goossens, K.: Architecture and optimal configuration of a real-time multi-channel memory controller. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp. 1307–1312 (2013). DOI 10.7873/DATE.2013.270
21. Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D.B., Harmon, M.G.: Bounding pipeline and instruction cache performance. IEEE Trans. Computers **48**(1), 53–70 (1999)

22. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. Tech. rep., AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013]
23. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and results of WCET tools. Proceedings of the IEEE **91**(7), 1038–1054 (2003)
24. Hepp, S., Brandner, F.: Splitting functions into single-entry regions. In: K.S. Chatha, R. Ernst, A. Raghunathan, R. Iyer (eds.) 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2014, Uttar Pradesh, India, October 12-17, 2014, pp. 17:1–17:10. ACM (2014). DOI 10.1145/2656106.2656128
25. Huber, B., Hepp, S., Schoeberl, M.: Scope-based method cache analysis. In: Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), pp. 73–82. Madrid, Spain (2014). DOI 10.4230/OASIcs.WCET.2014.73
26. Huber, B., Prokesch, D., Puschner, P.: Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In: Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES 2013), pp. 163–172. The Association for Computing Machinery (2013). DOI 10.1145/2499369.2465567
27. Huber, B., Puffitsch, W., Schoeberl, M.: WCET driven design space exploration of an object cache. In: Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010), pp. 26–35. ACM, New York, NY, USA (2010). DOI 10.1145/1850771.1850775
28. Hwu, W.M.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: An effective technique for vliw and superscalar compilation. The Journal of Supercomputing **7**(1), 229–248 (1993). DOI 10.1007/BF01205185
29. Jordan, A., Brandner, F., Schoeberl, M.: Static analysis of worst-case stack cache behavior. In: Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013), pp. 55–64. ACM, New York, NY, USA (2013). DOI 10.1145/2516821.2516828
30. Kasapaki, E., Schoeberl, M., Sørensen, R.B., Müller, C.T., Goossens, K., Sparsø, J.: Argo: A real-time network-on-chip architecture with an efficient GALS implementation. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **24**, 479–492 (2016). DOI 10.1109/TVLSI.2015.2405614
31. Kluge, F., Gerdes, M., Ungerer, T.: An operating system for safety-critical applications on manycore processors. In: 17th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2014, pp. 238–245. IEEE (2014)
32. Kluge, F., Schoeberl, M., Ungerer, T.: Support for the logical execution time model on a time-predictable multicore processor. In: 14th International Workshop on Real-Time Networks. ACM SIGBED Review, Toulouse, France (2016)
33. Lakis, E., Schoeberl, M.: An SDRAM controller for real-time systems. In: Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems (2013)
34. Lam, M.: Software pipelining: An effective scheduling technique for vliw machines. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, pp. 318–328. ACM, New York, NY, USA (1988). DOI 10.1145/53990.54022
35. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In: E.R. Altman (ed.) Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008), pp. 137–146. ACM, Atlanta, GA, USA (2008)
36. Lisper, B.: SWEET – a Tool for WCET Flow Analysis. In: B. Steffen (ed.) 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, pp. 482–485. Springer-Verlag (2014)
37. Liu, I.: Precision timed machines. Ph.D. thesis, EECS Department, University of California, Berkeley (2012)
38. Liu, I., Reineke, J., Broman, D., Zimmer, M., Lee, E.A.: A PRET microarchitecture implementation with repeatable timing and competitive performance. In: Proceedings of IEEE International Conference on Computer Design (ICCD 2012) (2012)

39. Liu, I., Reineke, J., Lee, E.A.: A PRET architecture supporting concurrent programs with composable timing properties. In: Signals, Systems and Computers, 2010 Conference Record of the Forty-Four Asilomar Conference on (2010)
40. Mälardalen Real-Time Research Center: WCET benchmarks. Available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html` (accessed 2009)
41. Metzlaff, S., Ungerer, T.: Replacement policies for a function-based instruction memory: A quantification of the impact on hardware complexity and wcet estimates. In: Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on, pp. 112 –121 (2012). DOI 10.1109/ECRTS.2012.22
42. Mische, J., Guliashvili, I., Uhrig, S., Ungerer, T.: How to enhance a superscalar processor to provide hard real-time capable in-order smt. In: 23rd International Conference on Architecture of Computing Systems (ARCS 2010), pp. 2–14. Springer, University of Augsburg, Germany (2010). DOI 10.1007/978-3-642-11950-7_2
43. Pitter, C., Schoeberl, M.: A real-time Java chip-multiprocessor. ACM Trans. Embed. Comput. Syst. **10**(1), 9:1–34 (2010). DOI 10.1145/1814539.1814548
44. Prokesch, D., Hepp, S., Puschner, P.P.: A Generator for Time-Predictable Code. In: IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC 2015, Auckland, New Zealand, 13-17 April, 2015, pp. 27–34. IEEE Computer Society (2015). DOI 10.1109/ISORC.2015.40
45. Prokesch, D., Huber, B., Puschner, P.: Towards Automated Generation of Time-Predictable Code. In: Int. Workshop on Worst-Case Execution Time Analysis, *OASIcs*, vol. 39, pp. 103–112. Schloss Dagstuhl (2014)
46. Puschner, P.: Experiments with WCET-oriented programming and the single-path architecture. In: Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (2005)
47. Puschner, P., Kirner, R., Huber, B., Prokesch, D.: Compiling for time predictability. In: F. Ortmeier, P. Daniel (eds.) Computer Safety, Reliability, and Security, *Lecture Notes in Computer Science*, vol. 7613, pp. 382–391. Springer Berlin / Heidelberg (2012)
48. Puschner, P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and WCET-analysis integration. In: 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013), pp. 33–40 (2013)
49. Rocha, A., Silva, C., Sørensen, R.B., Sparsø, J., Schoeberl, M.: Avionics applications on a time-predictable chip-multiprocessor. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2016), pp. 777–785. IEEE Computer Society (2016). DOI 10.1109/PDP.2016.36
50. Rochange, C., Sainrat, P.: Towards designing WCET-predictable processors. In: Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003, pp. 87–90 (2003)
51. Schoeberl, M.: A time predictable instruction cache for a Java processor. In: On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), *LNCS*, vol. 3292, pp. 371–382. Springer, Agia Napa, Cyprus (2004). DOI 10.1007/b102133
52. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture **54/1–2**, 265–286 (2008). DOI http://dx.doi.org/10.1016/j.sysarc.2007.06.001
53. Schoeberl, M.: Time-predictable cache organization. In: Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009), pp. 11–16. IEEE Computer Society, Tokyo, Japan (2009). DOI 10.1109/STFSSD.2009.10
54. Schoeberl, M.: Time-predictable computer architecture. EURASIP Journal on Embedded Systems **vol. 2009, Article ID 758480**, 17 pages (2009). DOI 10.1155/2009/758480
55. Schoeberl, M.: Is time predictability quantifiable? In: International Conference on Embedded Computer Systems (SAMOS 2012). IEEE, Samos, Greece (2012)
56. Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P., Rocha, A., Silva, C., Sparsø, J., Tocchi, A.: T-CREST: Time-predictable multi-core architecture

for embedded systems. Journal of Systems Architecture **61**(9), 449–471 (2015). DOI http://dx.doi.org/10.1016/j.sysarc.2015.04.002
57. Schoeberl, M., Brandner, F., Hepp, S., Puffitsch, W., Prokesch, D.: Patmos reference handbook. Tech. rep., Technical University of Denmark (2014)
58. Schoeberl, M., Brandner, F., Sparsø, J., Kasapaki, E.: A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In: Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS), pp. 152–160. IEEE, Lyngby, Denmark (2012). DOI 10.1109/NOCS.2012.25
59. Schoeberl, M., Chong, D.V., Puffitsch, W., Sparsø, J.: A time-predictable memory network-on-chip. In: Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014), pp. 53–62. Madrid, Spain (2014). DOI 10.4230/OASIcs.WCET.2014.53
60. Schoeberl, M., Huber, B., Puffitsch, W.: Data cache organization for accurate timing analysis. Real-Time Systems **49**(1), 1–28 (2013). DOI 10.1007/s11241-012-9159-8
61. Schoeberl, M., Puffitsch, W., Huber, B.: Towards time-predictable data caches for chip-multiprocessors. In: Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009), no. 5860 in LNCS, pp. 180–191. Springer (2009)
62. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: Towards a time-predictable dual-issue microprocessor: The Patmos approach. In: First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), pp. 11–20. Grenoble, France (2011)
63. Sparsø, J., Kasapaki, E., Schoeberl, M.: An area-efficient network interface for a TDM-based network-on-chip. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13, pp. 1044–1047. EDA Consortium, San Jose, CA, USA (2013)
64. Starke, R.A.: Design and evaluation of a vliw processor for real-time systems. Ph.D. thesis, Universidade Federal de Santa Catarina (2016)
65. Starke, R.A., Carminati, A., Oliveira, R.S.D.: Evaluating the design of a VLIW processor for real-time systems. ACM Trans. Embed. Comput. Syst. **15**(3), 46:1–46:26 (2016). DOI 10.1145/2889490
66. Thiele, L., Wilhelm, R.: Design for timing predictability. Real-Time Systems **28**(2-3), 157–177 (2004)
67. Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quiñones, E., Gerdes, M., Paolieri, M., Wolf, J.: Merasa: Multi-core execution of hard real-time applications supporting analysability. Micro, IEEE **30**(5), 66–75 (2010). DOI 10.1109/MM.2010.78
68. Whitham, J.: Real-time processor architectures for worst case execution time reduction. Ph.D. thesis, University of York (2008)
69. Whitham, J., Audsley, N.: Using trace scratchpads to reduce execution times in predictable real-time architectures. In: Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), pp. 305–316 (2008). DOI 10.1109/RTAS.2008.11
70. Whitham, J., Audsley, N.: Implementing time-predictable load and store operations. In: Proceedings of the International Conference on Embedded Software (EMSOFT 2009) (2009)
71. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Transactions on CAD of Integrated Circuits and Systems **28**(7), 966–978 (2009)
72. Ziccardi, M., Schoeberl, M., Vardanega, T.: A time-composable operating system for the Patmos processor. In: The 30th ACM/SIGAPP Symposium On Applied Computing, Embedded Systems Track. ACM Press, Salamanca, Spain. (2015)
73. Zimmer, M., Broman, D., Shaver, C., Lee, E.A.: FlexPRET: A processor platform for mixed-criticality systems. In: Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS). Berlin, Germany (2014)